A Type Theory with Partial Equivalence Relations as Types

Abhishek Anand  Mark Bickford  Robert L. Constable  Vincent Rahli

May 13, 2014
Stuart Allen’s Thesis

This work started with a careful reading of:

Stuart Allen’s PhD thesis [All87]:
A Non-Type-Theoretic Semantics for Type-Theoretic Language

It describes a semantics for Nuprl where types are defined as Partial Equivalence Relations on terms (the PER semantics).
Among others, Nuprl has the following types:

**Equality**: $a = b \in T$

**Dependent function**: $a: A \to B[a]$

**Dependent product**: $a : A \times B[a]$

**Intersection**: $\cap a : A. B[a]$

**Partial**: $\overline{\bar{A}}$

**Universe**: $\mathbb{U} ;$

**Subset**: $\{a : A \mid B[a]\}$

**Quotient**: $T//E$

where $E$ has to be an equivalence relation w.r.t. $T$. 
Stuart Allen's Thesis

In his thesis, the following page was misplaced:

forming an $a \in A$ such that $B_{E_{A}}$ is inhabited; two equal canonical members
are formed by forming $a, a' \in \{ z \in A \mid B_{z} \}$ such that $E_{A}(a, a')$ is inhabited.

The set type and quotient type constructor could have been unified in a
single constructor $\nu \in A / E_{A}$ which is the quotient except that, rather
than requiring the inhabitation of $E_{A}$ to be an equivalence relation, we
require only that it be transitive and symmetric over $A$, i.e., its restriction
to $A$ should be a partial equivalence relation. The equal members are the
members of $A$ that make $E_{A}$ inhabited. Thus, a type $a, a' \in A / E_{A}$ is
extensionally equal to $a, a' \in A / E_{A}$, and a type $\{ z \in A \mid B_{z} \}$ is extensionally
equal to $\{ z, z' \mid E_{A}(z, z') \}$.

We come now to Nuprl's treatment of assumptions. Nuprl uses one form
of judgement:

$$a_{1} : A_{1}, \ldots, a_{n} : A_{n} \Rightarrow \mathbb{N}$$

Let us start by considering Nuprl judgements with one assumption. The
meaning of $a : A \Rightarrow \mathbb{N}$ is that, for any $z$ and $a'$, if $z = a' : A$ then $T_{A_{1}, \ldots, A_{n}} - f(z) = f(a') = T_{A_{1}, \ldots, A_{n}}$. Notice that, rather than implying or
presupposing that $A$ is a type, the typehood of $A$ is part of the assumption
(since the typehood of $A$ is implied by $a \in A$). Thus, if $A$ cannot be
defined as a type, because it has no value, say, then we may infer for any $z_{1}, z_{2}$
and $f$ that $x = f(z_{1} ; z_{2})$. In contrast, we cannot infer $I_{z} : T_{A_{1}, \ldots, A_{n}}$ unless we
also know that $A$ is a type. Since we are discussing two forms of assumption,
it will be convenient to introduce a distinguishing nomenclature; there will be
no need to make the general application of the terminology precise. We shall
say an assumption $a : A$ is positive within the judgement that, by virtue of
that assumption, imply the typehood of $A$, and we shall say the assumption is
negative within the judgements in which the typehood of $A$ is a part of what
is being assumed. The assumption $a \in A$ is positive within $\{ z \in \mathbb{N} \mid a \in A \}$ and
negative within $\{ z \in \mathbb{N} \mid f(z) \in \mathbb{N} \}$. The use of negative assumptions allows one
to express the assumption that $a$ is a member of $A$ as a negative assumption
$a : \{ a \in A \mid \alpha \}$. A positive assumption of this form would be useless since for
$\{ a \in A \mid \alpha \}$ to be a type $A$ must be a type with member $a$.

Now we shall consider judgements that use two negative assumptions.
The meaning intended for judgements using more assumptions should be
clear in light of the explanation for two assumptions. A coarse reading, one

\[a_{1} : A_{1}, \ldots, a_{n} : A_{n} \Rightarrow \mathbb{N} \]
What does it say?

It suggests that the \textit{quotient} and \textit{subset} types could be replaced by a quotient-like type that only requires a partial equivalence relation.
Our Proposal

Here is our proposal—redefining Nuprl’s type theory around an extensional “Partial Equivalence Relation” type constructor that turns PERs into types.

The domain: the closed terms of Nuprl’s computation system.

**Base** is the type that contains all closed terms and whose equality $\sim$ is Howe’s computational equivalence relation [How89].
Our Proposal

Now, the **per** type constructor:

- \( \text{per}(R) \) is a type if \( R \) is a PER on \( \text{Base} \).
- \( a = b \in \text{per}(R) \) if \( R \ a \ b \).
- \( \text{per}(R_1) = \text{per}(R_2) \in \mathbb{U}_i \) if \( R_1 \) and \( R_2 \) are equivalent relations.

We’ll need universes as well.

Our type theory now has: Base, \( \mathbb{U}_i \), per.
Our Proposal

per types are now part of our implementation of Nuprl in Coq [AR14]. We verified:

\[
H \vdash \text{per}(R) = \text{per}(R') \in \text{Type} \quad \text{BY [pertypeEquality]}
\]
\[
H, x : \text{Base}, y : \text{Base} \vdash R \times y \in \text{Type}
\]
\[
H, x : \text{Base}, y : \text{Base} \vdash R' \times y \in \text{Type}
\]
\[
H, x : \text{Base}, y : \text{Base}, z : R \times y \vdash R' \times y
\]
\[
H, x : \text{Base}, y : \text{Base}, z : R' \times y \vdash R \times y
\]
\[
H, x : \text{Base}, y : \text{Base}, z : R \times y \vdash R \times y \quad \text{BY [ext e]}
\]
\[
H, x : t_1 = t_2 \in \text{per}(R) \vdash C \quad \text{BY [pertypeElimination]}
\]
\[
H, x : t_1 = t_2 \in \text{per}(R), [y : R \ t_1 \ t_2] \vdash C \quad \text{BY [ext e]}
\]
\[
H \vdash t_1 = t_2 \in \text{per}(R) \quad \text{BY [pertypeMemberEquality]}
\]
\[
H \vdash \text{per}(R) \in \text{Type}
\]
\[
H \vdash R \ t_1 \ t_2
\]
\[
H \vdash t_1 \in \text{Base}
\]
\[
H \vdash t_2 \in \text{Base}
\]
Examples

Let us start with simple examples:

\[
\text{Void} = \text{per}(\lambda -, .1 \preceq 0)
\]

\[
\text{Unit} = \text{per}(\lambda -, .0 \preceq 0)
\]

These use \(\preceq\), Howe’s computational approximation relation [How89].

Our type theory now has: Base, \(\mathbb{U}_i\), per, \(\preceq\).
Examples

Integers:

\[ \mathbb{Z} = \text{per}(\lambda a. \lambda b. a \sim b \sqcap \uparrow(\text{isint}(a, \text{tt}, \text{ff}))) \]

where

\[ A \sqcap B = \sqcap x: \text{Base}. \sqcap y: \text{halts}(x). \text{isaxiom}(x, A, B) \]

\[ \uparrow(a) = \text{tt} \preceq a \]

\[ \text{halts}(t) = A x \preceq (\text{let } x := t \text{ in } A x) \]

Our type theory now has: \text{Base, } \mathbb{U};, \text{per, } \preceq, \sim, \sqcap.
Examples

Quotient types:

\[ T // E = \text{per}(\lambda x, y. (x \in T) \cap (y \in T) \cap (E \times y)) \]

This is the definition we are using in Nuprl now—no longer a primitive.

The partial type constructor is a quotient type—no longer a primitive.

Our type theory now has: Base, \( \mathbb{U} \), per, \( \leq \), ~, \( \cap \), \( \_ = \_ \in \_ \).
What about the subset type?

\[ \{ a : A \mid B[a] \} = \text{per}(\lambda x, y. (x = y \in A) \cap B[x]) \]
Examples

What about the subset type?

\[ \{ a : A \mid B[a] \} = \text{per}(\lambda x, y. (x = y \in A) \cap B[x]) \]

This does not work!

We do not get that \(B\) is functional over \(A\).
one solution—annotate families with levels:

\[
\{a : A \mid B[a]\}; = \text{per}(\lambda x, y. (x = y \in A) \cap B[x] \cap \text{Fam}(A, B, i))
\]

where

\[
\text{Fam}(A, B, i) = \cap a, b: A. (B[a] = B[b] \in \mathbb{U}_i)
\]

**One drawback:** the annotations.
another solution—introduce a type of type equalities \((T = U)\):

\[
\{ a : A \mid B[a] \} = \text{per}(\lambda x, y. (x = y \in A) \sqcap B[x] \sqcap \text{Fam}(A, B))
\]

where

\[
\text{Fam}(A, B) = \sqcap a, b : A. (B[a] = B[b])
\]

This requires a more intensional version of our \text{per} type.
Examples

Using this method, we can also define the other type families such as: dependent functions, dependent products, …

Both \( \text{per} \) and its intensional version are part of our implementation of Nuprl in Coq [AR14].

We proved, e.g., that the elimination rule for the \( \text{per} \) version of our function type is valid.
Inductive types

We saw how to build inductive types in yesterday's talk.

- Algebraic datatypes: \( \{ t : coDT \mid \text{halts}(\text{size}(t)) \} \).
- Inductive types using Bar Induction.
Conclusion

❖ Conciseness
  ▶ A small core of primitive types.
  ▶ Simple rules.

❖ Flexibility
  ▶ Lets user define even more types.
  ▶ No need to modify/update the meta-theory.

❖ Practicality?
  ▶ We’re already using it.
  ▶ We’re still experimenting with the intensional per type.
Stuart F. Allen.
A Non-Type-Theoretic Semantics for Type-Theoretic Language.

Abhishek Anand and Vincent Rahli.
Towards a formally verified proof assistant.
Accepted to ITP 2014, 2014.

Douglas J. Howe.
Equality in lazy computation systems.